

Assembly Language and Computer Architecture Using C++ and Java

8/23/2010

Chapter 1

Number Systems

Would you trust a medical doctor who did not know the location of the human appendix?

Not likely.

A doctor must know the structure and operation of the human body.

Similarly, a computer expert must know the structure and operation of computers.

To provide a clear, concrete, and thorough view of the structure and operation of computers is our objective.

Positional number systems

- Decimal: base 10
- Binary: base 2
- Hexadecimal: base 16

Decimal

- Ten symbols: 0, 1, 2, ..., 9
- Weights change by a factor of 10 from one position to the next.
- Shifting the decimal point changes the value by a factor of 10 for each position shifted.
- Each symbol is called a “digit”

25.4 decimal

$$\begin{array}{r} 25.4 \\ \hline 10 \quad 1 \quad \frac{1}{10} \end{array}$$

weights

25.4 decimal =

$$2 \times 10 + 5 \times 1 + 4 \times \frac{1}{10}$$

Binary

- Two symbols: 0 and 1
- Weights change by a factor of 2 from one position to the next.
- Shifting the binary point changes the value by a factor of 2 for each position shifted.
- Each symbol is called a “bit”.

1011.1 binary

$$\begin{array}{cccccc} 1 & 0 & 1 & 1 & \cdot & 1 \\ \hline 8 & 4 & 2 & 1 & & \frac{1}{2} \end{array} \quad \text{weights (in decimal)}$$

and its value in decimal is

$$1 \times 8 + 0 \times 4 + 1 \times 2 + 1 \times 1 + 1 \times \frac{1}{2} = 11.5$$

Most and least significant bits

0111010110010111

MSB



LSB



Hexadecimal

- Sixteen symbols: 0, 1, 2, ..., 9, A, B, C, D, E, F
- Weights change by a factor of 16 from one position to the next.
- Shifting the hexadecimal point changes the value by a factor of 16 for each position shifted.
- Each symbol is called a “hex digit”
- A = 10 decimal B = 11 decimal
- C = 12 decimal D = 13 decimal
- E = 14 decimal F = 15 decimal

1CB.8 hex

$$\begin{array}{ccccccc} 1 & C & B & . & 8 & & \\ \hline 256 & 16 & 1 & & \frac{1}{16} & & \end{array} \quad \text{weights (in decimal)}$$

and its value is

$$1 \times 256 + C \times 16 + B \times 1 + 8 \times \frac{1}{16}$$

Substituting the decimal equivalents for the symbols B (11 decimal) and C (12 decimal), we get an all-decimal expression from which we can compute its decimal value:

$$1 \times 256 + 12 \times 16 + 11 \times 1 + 8 \times \frac{1}{16} = 459.5$$

Memorize this table

FIGURE 1.1

Decimal	Binary	Hexadecimal
0	0000	0
1	0001	1
2	0010	2
3	0011	3
4	0100	4
5	0101	5
6	0110	6
7	0111	7
8	1000	8
9	1001	9
10	1010	A
11	1011	B
12	1100	C
13	1101	D
14	1110	E
15	1111	F

Complement of a symbol s in a base n system

$$(n - 1) - s$$

Binary: Complement of 1 is $1 - 1 = 0$

Decimal: Complement of 3 = $9 - 3 = 6$

Hex: Complement of 4 = $F - 4 = B$

Byte

- [illegible]

FIGURE 1.2**Distinct Binary Numbers**

Number of Bits	Decimal			Hex
4	$2^4 =$		16 =	10
7	$2^7 =$		128 =	80
8	$2^8 =$		256 =	100
10	$2^{10} =$	1K =	1,024 =	400
12	$2^{12} =$	4K =	4,096 =	1000
15	$2^{15} =$	32K =	32,768 =	8000
16	$2^{16} =$	64K =	65,536 =	10,000
20	$2^{20} =$	1M =	1,048,576 =	100,000
30	$2^{30} =$	1G =	1,073,741,824 =	40,000,000

Arithmetic with positional numbers

Rules are essentially the same for all bases

Arithmetic in decimal

- In addition, whenever a column sum is greater than or equal to the number base, a carry is added to the next column. The column result is the rightmost symbol of the column sum; the carry is the left symbol(s) of the column sum. For example, in the decimal addition,

$$\begin{array}{r} 1 \leftarrow \text{carry} \\ 28 \\ + 39 \\ \hline 67 \end{array}$$

the right column sum is 17. Thus, the result digit in the right column is 7 with a carry of 1 into the next column.

- In subtraction, a borrow into a column is equal to the number base. For example, in the decimal subtraction

$$\begin{array}{r} 34 \\ - 8 \\ \hline 26 \end{array}$$

we borrow 1 from the 3 in the left column. This borrow is worth 10 in the right column. Thus, we subtract 8 from the sum of 10 (the borrow) and 4 to get 6. We borrow whenever the bottom symbol in a column is greater than the effective top symbol (the top symbol less any borrows from it).

- A borrow from 0 propagates to the left. For example, in the subtraction

$$\begin{array}{r} 3000 \\ - 2 \\ \hline 2998 \end{array}$$

Adding in binary

$$\begin{array}{r} 111 \quad \leftarrow \text{carries} \\ 01100 \\ + 11110 \\ \hline 101010 \end{array}$$

Subtraction in binary

$$\begin{array}{r} 101 \\ - 011 \\ \hline 010 \end{array}$$

2 decimal ← borrow into second column

Addition in hex

$$\begin{array}{r} 1 \leftarrow \text{carry} \\ \text{A9} \\ + 19 \\ \hline \text{C2} \end{array}$$

Subtraction in hex

16 decimal ← borrow into right column

$$\begin{array}{r} \text{A5} \\ + \text{2B} \\ \hline \text{7A} \end{array}$$

Converting to base n

- Integer part: Repeatedly divide by n .
Remainders are the digits of the base n number.
- Fractional part: Repeatedly multiply by n .
Whole parts are the digits of the base n number.

Each remainder is a digit

	remainders	
$\begin{array}{r} 0 \\ 10 \overline{) 9} \end{array}$	9	
$\begin{array}{r} 0 \\ 10 \overline{) 90} \end{array}$	0	
$\begin{array}{r} 0 \\ 10 \overline{) 905} \end{array}$	5	

← start with this division and work up

↑

Convert 11 decimal to binary

	remainders	
0	1	
2) <u>1</u>	0	
2) <u>2</u>	1	
2) <u>5</u>	1	
2) <u>11</u>		← start with this division and work up

11 decimal = 1011 binary

Convert 30 decimal to hex

remainders

0	1	
$16 \overline{) 1}$	$14 = E$	\uparrow
$16 \overline{) 30}$		\leftarrow start with this division and work up

30 decimal = 1E hex

Each whole part is a digit (strip off whole part of product after each multiplication by 10)

$$\begin{array}{r} .43 \\ \times 10 \\ \hline 1^{\text{st}} \text{ digit} \longrightarrow 4. \quad 3 \\ \\ \times 10 \\ \hline 2^{\text{nd}} \text{ digit} \longrightarrow 3. \quad 0 \end{array}$$

Convert .6875 decimal to binary

whole
parts

$$\begin{array}{r} .6875 \\ \times 2 \\ \hline 1 \quad .3750 \\ \times 2 \\ \hline 0 \quad .7500 \\ \times 2 \\ \hline 1 \quad .5000 \\ \times 2 \\ \hline 1 \quad .0000 \end{array}$$

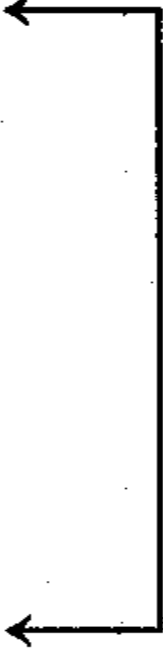
← start with this multiplication and work down



.6875 decimal = .1011 binary

Repeating fraction

whole
parts

	$\times \begin{array}{r} .1 \\ 2 \\ \hline \end{array}$	
0	$\begin{array}{r} .2 \\ \hline \end{array}$	
	$\times \begin{array}{r} .2 \\ 2 \\ \hline \end{array}$	
0	$\begin{array}{r} .4 \\ \hline \end{array}$	
	$\times \begin{array}{r} .2 \\ 2 \\ \hline \end{array}$	
0	$\begin{array}{r} .8 \\ \hline \end{array}$	
	$\times \begin{array}{r} .2 \\ 2 \\ \hline \end{array}$.2 repeats
1	$\begin{array}{r} .6 \\ \hline \end{array}$	
	$\times \begin{array}{r} .2 \\ 2 \\ \hline \end{array}$	
1	$\begin{array}{r} .2 \\ \hline \end{array}$	

.1 decimal = .0 0011 0011 ... binary

Converting between binary and hex

- Very easy to do
- The ease of conversion is the reason why hex is often used as a shorthand representation of binary.
- To do conversion, you need to know the binary-hex equivalents for the numbers 0 to 15.

Binary to hex

0111	1010	1100	•	1110
↓	↓	↓		↓
7	A	C	•	E

Hex to binary

2	A	D	•	B	E
↓	↓	↓		↓	↓
0010	1010	1101	•	1011	1110

Horner's method

Efficient technique for converting
to a new number base

Inefficient evaluation of 2CD5

$$2 \times 16^3 + 12 \times 16^2 + 13 \times 16 + 5$$

A straightforward, but inefficient, way to evaluate the above expression is to

1. Compute 16^3 (two multiplications) and multiply by 2.
2. Compute 16^2 (one multiplication) and multiply by 12.
3. Multiply 13 by 16.
4. Add the preceding three products along with 5, the least significant digit.

Use Horner's rule to convert 2CD5

2	
$\times \quad 16$	multiply MSD (2) by 16
<hr/> 32	product
$+\quad 12$	add next digit (C)
<hr/> 44	
$\times \quad 16$	multiply result by 16
<hr/> 264	
44	
<hr/> 704	product
$+\quad 13$	add next digit (D)
<hr/> 717	
$\times \quad 16$	multiply result by 16
<hr/> 4302	
717	
<hr/> 11472	product
$+\quad 5$	add LSD (5)
<hr/> 11477	final result

Horner's rule written horizontally

$$((2 \times 16 + 12) \times 16 + 13) \times 16 + 5$$



2 multiplied by 16 three times

Convert 38 to binary using Horner's rule

	0011	= 3 decimal
×	1010	= 10 decimal
<hr/>		
	0000	
	0011	
	0000	
	0011	
<hr/>		
	0011110	product
+	1000	= 8 decimal
<hr/>		
	0100110	= 38 decimal

Signed binary numbers

- Sign-Magnitude
- Two's Complement
- One's Complement
- Excess-n

Sign-magnitude

sign bit
↓
10000000000000101 = -5
└──────────────────┘
magnitude

sign bit
↓
00000000000000101 = +5
└──────────────────┘
magnitude

Sign-magnitude representation not good for computers

Adding sign-magnitude numbers requires sign analysis:

Like signs, then add magnitudes

Unlike signs, then subtract magnitudes

Two's complement

The two's complement of the number M is the number which yields 0 when added to M .

Note this behavior

carry
discarded

$$\begin{array}{r} 1111111111111111 \\ + \quad \quad \quad 1 \\ \hline 0000000000000000 \end{array}$$

16 bit number containing all 1's
add 1
get all zeros

Simply flipping the bits does not
yield the two's complement.

$$\begin{array}{r} 0000000000000000101 = +5 \\ + 111111111111111010 = +5 \text{ flipped} \\ \hline 111111111111111111 \end{array} \quad \begin{array}{l} \\ \\ \text{should be zero} \end{array}$$

Getting the two's complement

- Simply flipping the bits does not yield the two's complement—it yields all 1's.
- But adding 1 to all 1's yields all 0's.
- So flipping the bits **and** adding 1 should yield the two's complement.

Flip bits and add 1

$$111111111111010 = +5 \text{ flipped}$$

$$\begin{array}{r} + 1 \\ \hline \end{array}$$

$$111111111111011 = +5 \text{ flipped} + 1$$

Now let's see if this new value, +5 flipped + 1, yields zero when added to +5:

carry



$$0000000000000101 = +5$$

$$\begin{array}{r} + 111111111111011 = +5 \text{ flipped} + 1 \\ \hline \end{array}$$

$$0000000000000000$$

Two's complement of 1

We can find the two's complement of any number using the procedure that we used on +5: Flip all the bits and then add one. For example, the computation of the two's complement of

$$0000000000000001 = +1$$

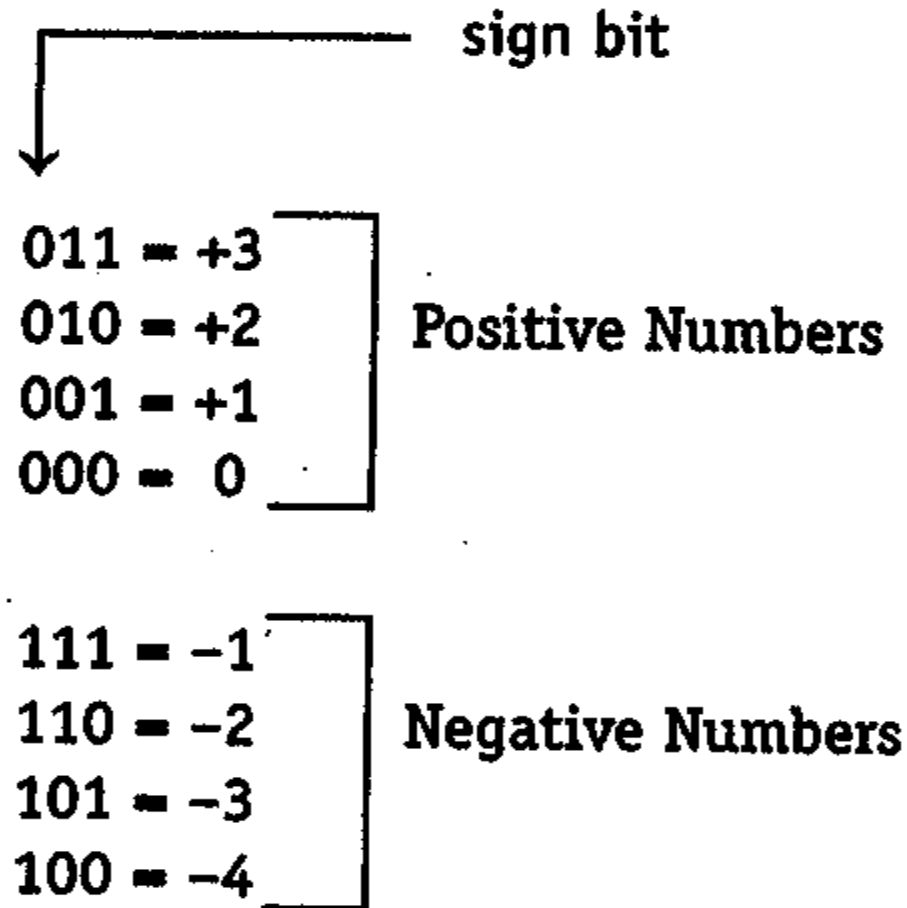
is:

$$1111111111111110 = +1 \text{ flipped}$$

$$\begin{array}{r} + 1 \\ \hline 1111111111111111 = -1 \end{array}$$

3-bit two's complement numbers

FIGURE 1.3



Terminology

1. To *complement a bit* means to flip the bit.
2. To *complement a number* means to take its two's complement (i.e., flip its bits and add one).
3. To *bitwise complement a number* means to flip each of its bits but *not* add one. The bitwise complement of a number is sometimes called the *one's complement* of the number (see the next section).

Positive result is in true form.
Negative result in complement form.

$$\begin{array}{r} 00000000000000101 = +5 \\ + 1111111111111111 = -1 \\ \hline 00000000000000100 = +4 \end{array}$$

The result is +4 in true form. Now let's add +2 and -5.

$$\begin{array}{r} 0000000000000010 = +2 \\ + 1111111111111011 = -5 \\ \hline 1111111111111101 = -3 \end{array}$$

One's complement

- Like two's complement, but don't add 1
- Requires addition of end-around carry
- $000000000000000001 = +1$
- $111111111111111110 = -1$

Adding one's complement numbers

carry out

$$\begin{array}{r} 1111111111111110 = -1 \text{ in one's complement} \\ + 1111111111111110 = -1 \text{ in one's complement} \\ \hline 1111111111111100 \quad \text{intermediate sum} \\ \boxed{1} \xrightarrow{\hspace{1.5cm}} \quad + 1 \quad \text{end-around carry} \\ \hline 111111111111101 = -2 \text{ final sum} \end{array}$$

One's complement has two
representations of 0:

0000000000000000

and

1111111111111111

This feature is not good for
computers.

Excess-n representation

- n is added to a number to get its excess- n representation.
- n is subtracted from an excess- n number to get its value.

FIGURE 1.4

Value	Excess-4	Two's Complement
-4	000	100
-3	001	101
-2	010	110
-1	011	111
0	100	000
1	101	001
2	110	010
3	111	011

When adding two excess-n numbers, must subtract n

$$\begin{array}{rcl} & 101 & = \text{excess-4 representation of 1} \\ + & 110 & = \text{excess-4 representation of 2} \\ \hline & 1011 & = \text{intermediate sum} \\ - & 0100 & = 4 \\ \hline & 111 & = \text{correct sum} \end{array}$$

Subtracting by two's complement addition: subtracting +3 from 5

$$\begin{array}{r} 0000000000000000101 = +5 \\ + 111111111111111101 = -3 \\ \hline 0000000000000000010 = +2 \end{array}$$

Use two's complement to subtract unsigned numbers

$$\begin{array}{r} 0000000000000101 = 5 \\ - 0000000000000011 = 3 \\ \hline \end{array}$$

where both numbers are unsigned, we complement the bottom number and then add it to the top number, exactly as we did for signed numbers:

$$\begin{array}{r} 0000000000000101 = 5 \\ + 111111111111101 = -3 \\ \hline 000000000000010 = 2 \end{array}$$

Computers use the two's complement technique to subtract *both* signed and unsigned numbers.

Range of unsigned numbers

- n bits means 2^n patterns
- Thus, range is 0 to $2^n - 1$
- $n = 8$, range is 0 to $2^8 - 1$ (0 to 255)

Range of two's complement numbers

- Positive numbers have msb = 0. Thus, there are $n - 1$ bits that can be set.
- Positive range is 0 to $2^{n-1} - 1$
- Negative numbers have msb = 1. Thus, there are $n - 1$ bits that can be set
- Negative range is -2^{n-1} to -1 .
- Total range: -2^{n-1} to $2^{n-1} - 1$.

FIGURE 1.5**Ranges of Binary Numbers**

Number of Bits	Unsigned	Two's Complement
8	0 to 255	-128 to 127
12	0 to 4095	-2048 to 2047
16	0 to 64K - 1	-32,768 to 32,767
20	0 to 1M - 1	-512K to 512K - 1
32	0 to 4G - 1	-2G to 2G - 1
n	0 to $2^n - 1$	-2^{n-1} to $2^{n-1} - 1$

Danger in using C++ char to represent signed number

- When extending a variable of type **char** to a longer format, extension may be signed number extension **OR** unsigned number extension.
- Type of extension depends on the compiler.

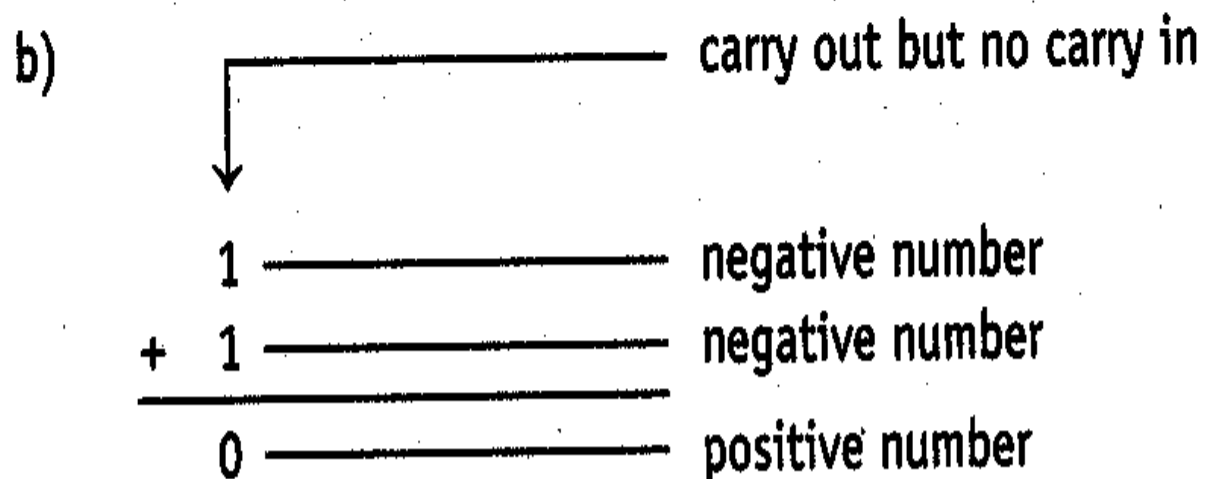
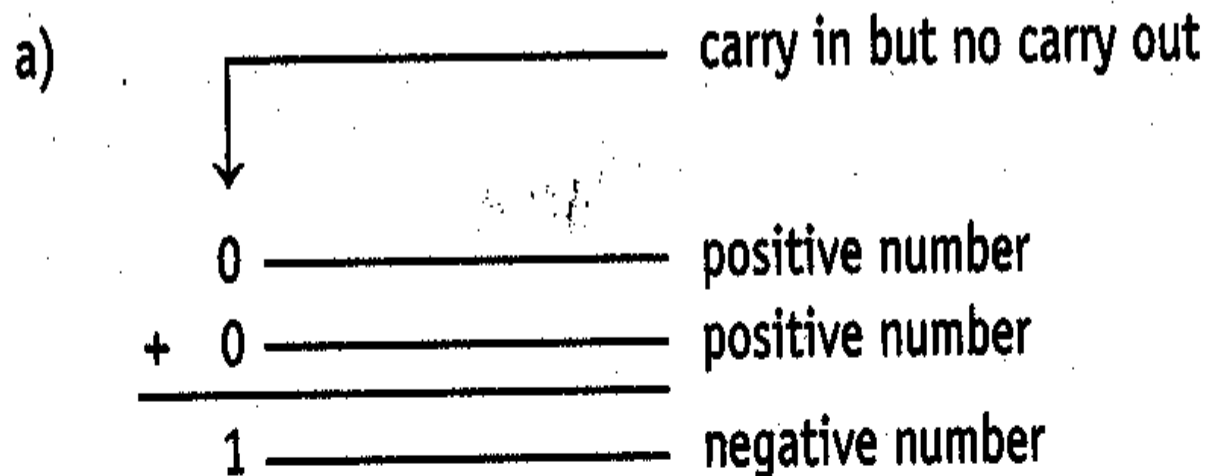
FIGURE 1.6

```
#include <iostream>
using namespace std;
int main()
{
    char c;
    c = -1;                // -1 is truncated to fit into c
    if (c == -1)           // promote c to int
        cout << "equal\n";
    else
        cout << "not equal\n";
    return 0;
}
```

Signed overflow

- Overflow never occurs when adding numbers of unlike signs.
- Overflow is indicated if the carry into the msb column differs from the carry out of the msb column.

FIGURE 1.7 Overflow Cases:



No-Overflow Cases:

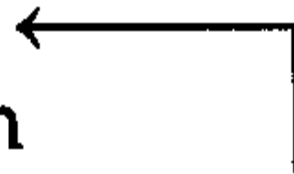
- c)
- | | | | |
|-------|---|-------|------------------------------|
| | | | no carry in and no carry out |
| | ↓ | | |
| | 0 | _____ | positive number |
| + | 0 | _____ | positive number |
| <hr/> | | | |
| | 0 | _____ | positive number |
- d)
- | | | | |
|-------|---|-------|------------------------|
| | | | carry in and carry out |
| | ↓ | | |
| | 1 | _____ | negative number |
| + | 1 | _____ | negative number |
| <hr/> | | | |
| | 1 | _____ | negative number |
- e)
- | | | | |
|-------|---|-------|------------------------|
| | | | carry in and carry out |
| | ↓ | | |
| | 0 | _____ | positive number |
| + | 1 | _____ | negative number |
| <hr/> | | | |
| | 0 | _____ | positive number |
- f)
- | | | | |
|-------|---|-------|------------------------------|
| | | | no carry in and no carry out |
| | ↓ | | |
| | 0 | _____ | positive number |
| + | 1 | _____ | negative number |
| <hr/> | | | |
| | 1 | _____ | negative number |

Flags

- **V** is the signed overflow flag
- **S** is the sign flag
- **C** is the carry flag

Unsigned overflow on an addition indicated by a carry out

carry out of
leftmost column



$$\begin{array}{r} 1111111111111111 = 65,535 \\ + 0000000000000001 = 1 \\ \hline 0000000000000000 \end{array}$$

Another type of unsigned overflow.
Indicated by a borrow into msb on
a subtraction.

$$\begin{array}{r} 000000000000000001 = 1 \\ - 000000000000000010 = 2 \\ \hline 1111111111111111 = 65,535 \end{array}$$

Unsigned overflow during a subtraction

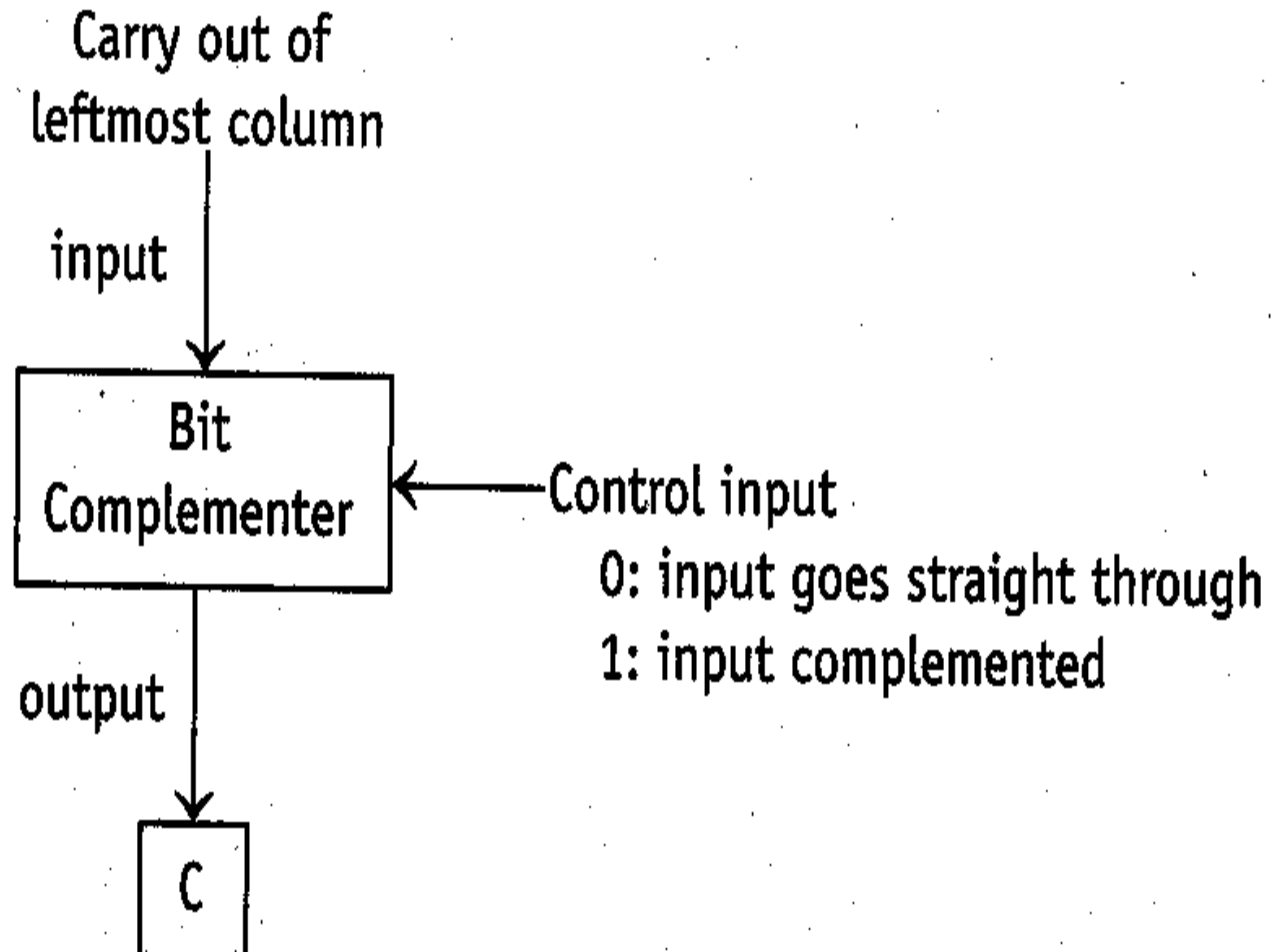
- Indicated by a borrow into the msb.
- But computers do **not** subtract using the borrow technique.
- Computers subtract using the two's complement technique. Carry out occurs if and only if a borrow in would **not** have occurred had the borrow technique been used.

Unsigned overflow test

- A borrow in occurs with the borrow technique if and only if a carry out does **not** occur with two's complement technique.
- On an addition, set the carry flag to the carry out. On subtraction, set the carry flag to the complement of the carry out. Then the carry flag functions as carry/borrow flag.
- Then for either addition or subtraction, a 1 in the carry flag indicates unsigned overflow.

Flip carry out on a subtraction

FIGURE 1.8



Analyzing two's complement numbers

2^n is an $n + 1$ bit number. For example,

$$2^4 = 10000$$

$2^n - 1$ is an n -bit number containing all 1's.

For example,

$$2^4 - 1 = 1111$$

Analyzing two's complement numbers

To flip an n-bit number, first subtract from all 1's:

$$\begin{array}{r} 1111 \\ - 0101 \\ \hline 1010 \quad (0101 \text{ flipped}) \end{array}$$

All 1's is $2^n - 1$. Thus, the n-bit **two's complement** of x is $2^n - 1 - x + 1 = 2^n - x$

Analyzing two's complement numbers

To add complement of x and x :

$$\begin{array}{r} 2^n - x \\ + x \\ \hline 2^n + 0 \end{array}$$

Carry out of leftmost column



Analyzing two's complement numbers

In mathematics, adding two positives yields a positive; adding two negatives yields a negative; adding a positive and a negative yields either a positive or negative depending on which of the two numbers added has the larger magnitude.

Two's complement numbers behave in the same way.

Analyzing two's complement numbers

Adding two positive two's complement numbers:

$$\begin{array}{r} a \\ b \\ \hline a + b \end{array}$$

The n-bit result equals the true value of $a + b$ as long as overflow does not occur.

Analyzing two's complement numbers

Adding two negative two's complement numbers:

$$2^n - c \quad (\text{two's complement of } c)$$

$$2^n - d \quad (\text{two's complement of } d)$$

$$2^n + 2^n - (c + d) \quad (\text{two's complement of } c+d)$$

↑
Lost

Notice that the sum of two negatives is negative.

Analyzing two's complement numbers


Adding a positive and a negative two's complement number, $a < d$:

$$\begin{array}{r} a \\ 2^n - d \quad (\text{two's complement of } d) \\ \hline 2^n - (d - a) \quad (\text{two's complement of } d - a) \end{array}$$

Notice the result is negative.

Analyzing two's complement numbers

Adding a positive and a negative two's complement number, $a \geq c$

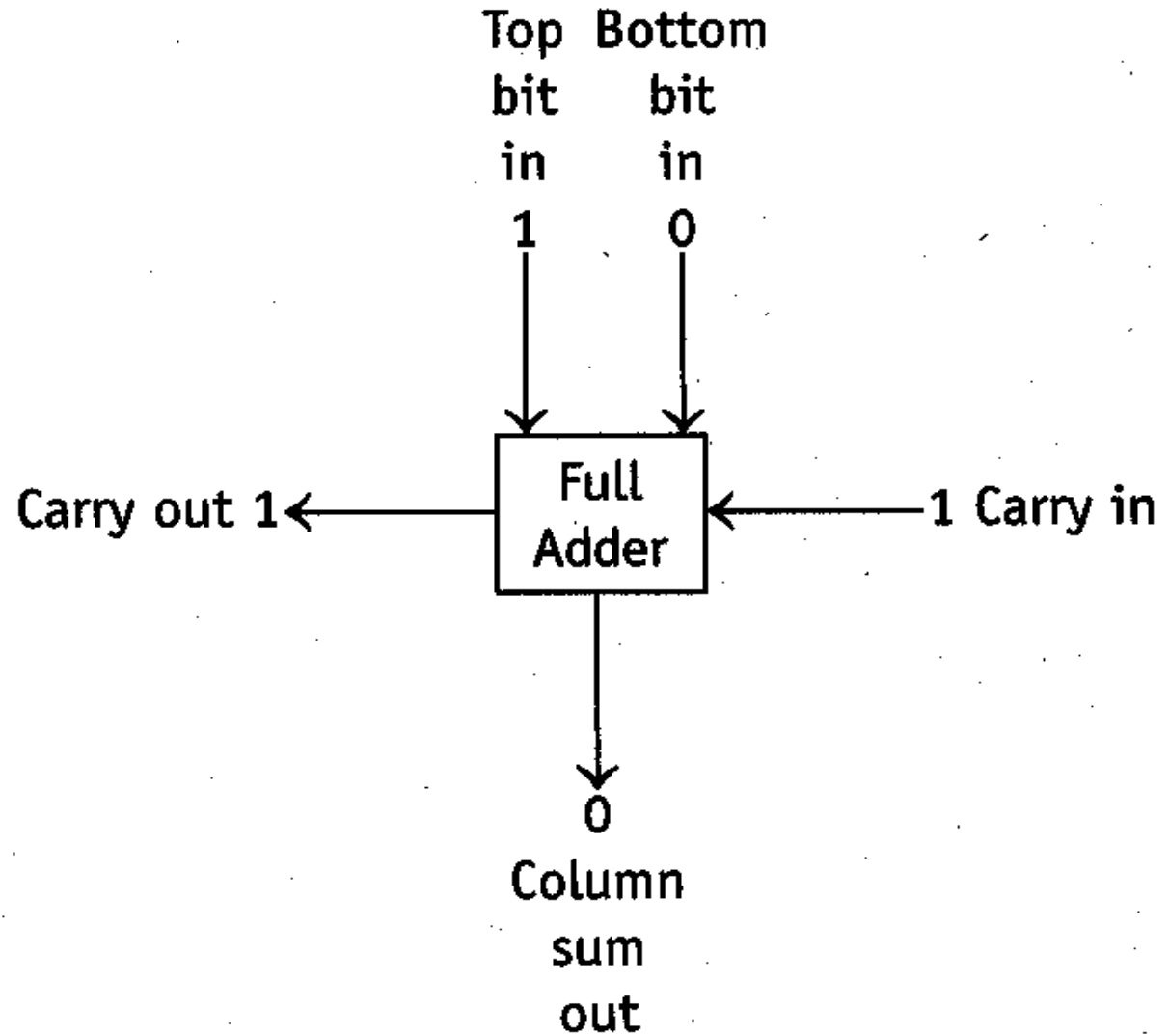
$$\begin{array}{r} a \\ 2^n - c \quad (\text{two's complement of } c) \\ \hline 2^n + a - c \end{array}$$


Lost

Notice the result is positive.

Full Adder

FIGURE 1.10



Full Adder is a circuit that adds one column

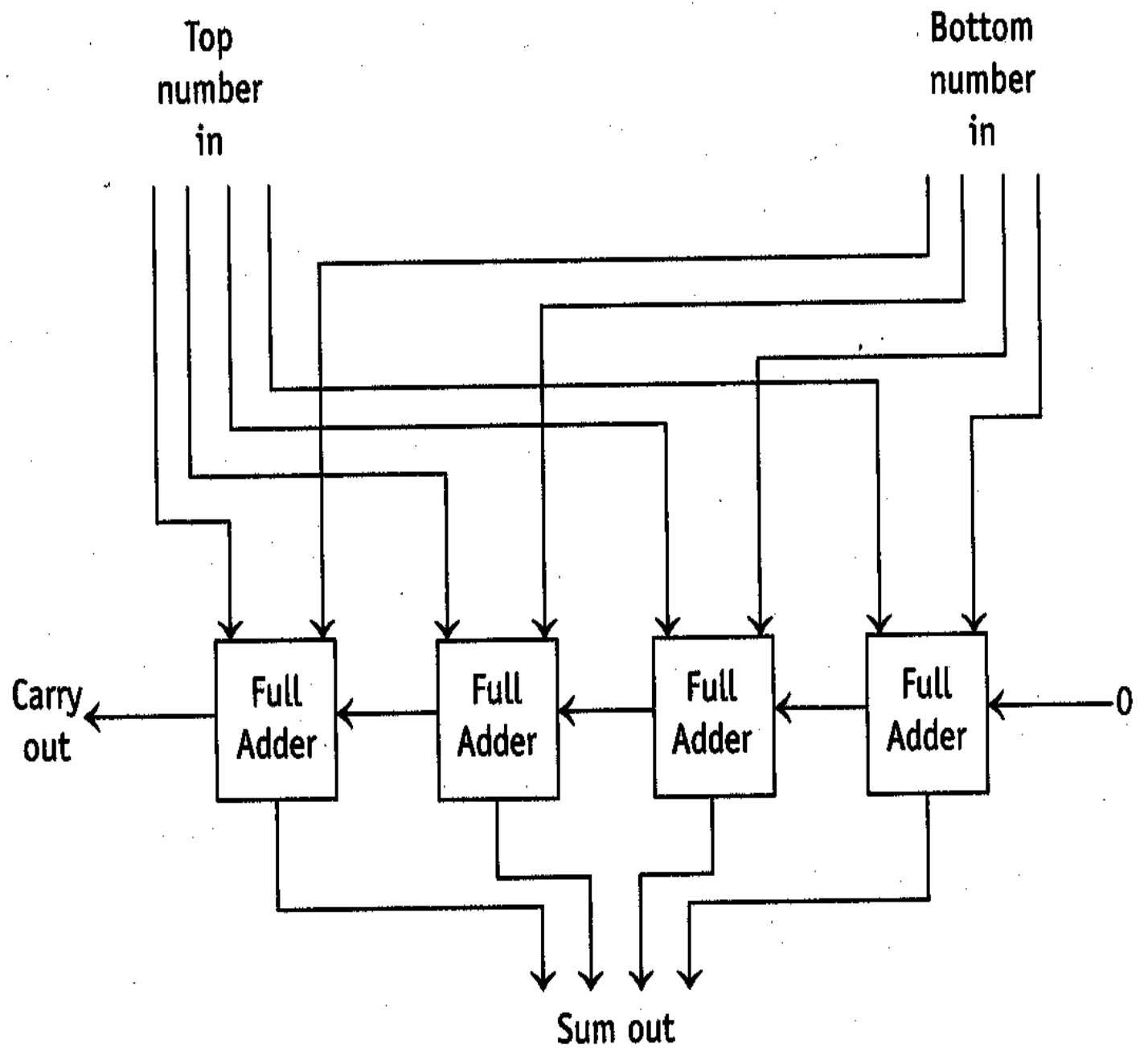
1 carry in from the right

1

+ 0

0 with a 1 carry to the left

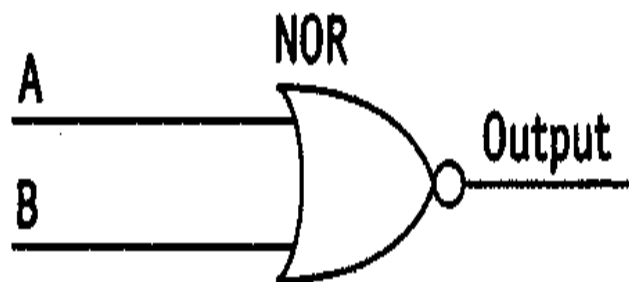
FIGURE 1.11



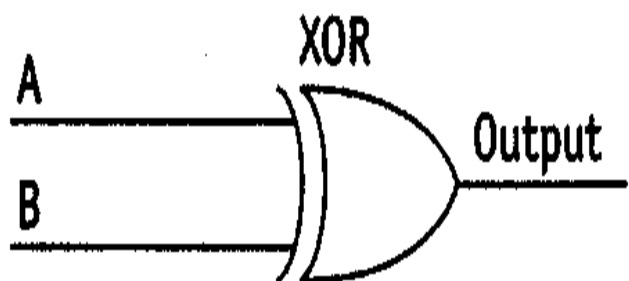
Gates

Simple circuits usually with two input lines and one output line, each carrying one bit. The output at any time depends on the inputs at that time.

FIGURE 1.12



A	B	Output
0	0	1
0	1	0
1	0	0
1	1	0



A	B	Output
0	0	0
0	1	1
1	0	1
1	1	0

XOR gate

Complementing gate

Complementing action of the XOR gate

$$\text{data XOR } 0 = \text{data}$$

$$\text{data XOR } 1 = \overline{\text{data}}$$

FIGURE 1.13

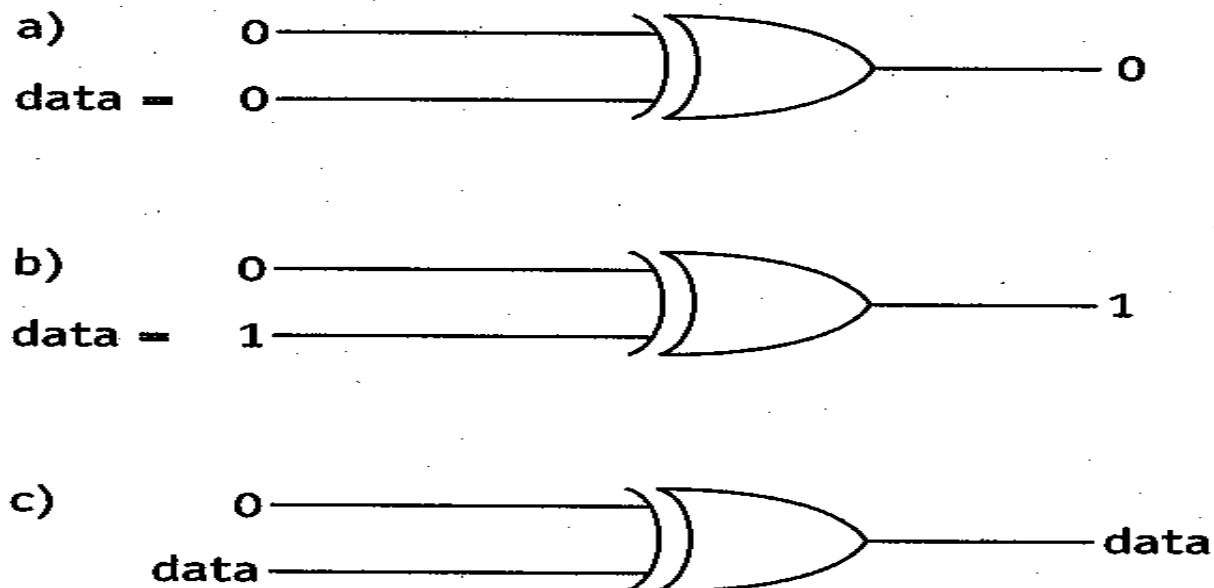
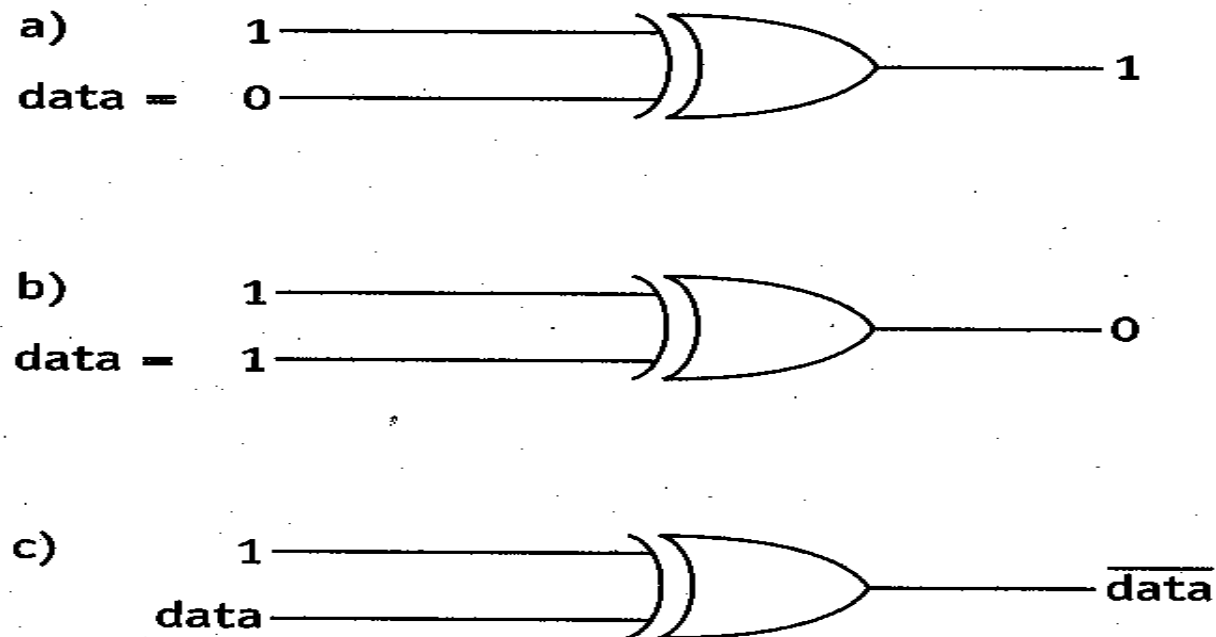


FIGURE 1.14

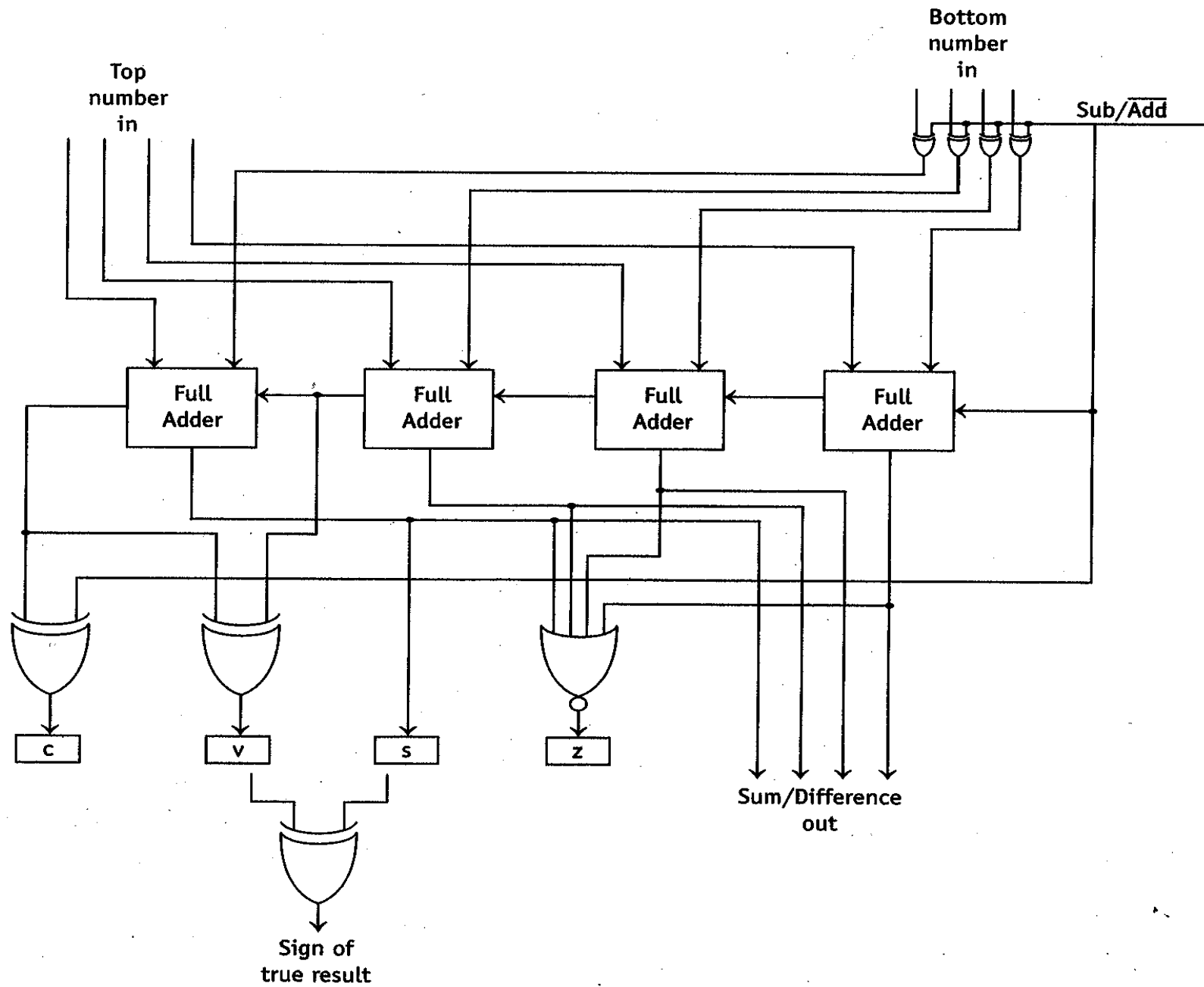


NOR gate

Zero-detecting gate
Outputs 1 if all inputs are 0

FIGURE 1.15

Adder/Subtractor



Subtracting a number from itself

- Using the borrow technique, a borrow would never occur.
- Thus, using the two's complement technique, a carry out should **ALWAYS** occur.

A carry out should always occur
when subtracting a number from
itself.

carry out ←

$$\begin{array}{r} 0000000000000101 = +5 \\ + 111111111111011 = -5 \\ \hline 0000000000000000 \end{array}$$

A carry out, indeed, occurs. For another test, let's subtract 0 from 0 (by adding the two's complement of 0). Because the two's complement of 0 is also 0, we get

$$\begin{array}{r} 0000000000000000 = 0 \\ + 0000000000000000 = 0 \neq \text{two's complement of 0} \\ \hline 0000000000000000 \end{array}$$

Subtracting 0 from 0 is NOT an exception

carry out ←

$$\begin{array}{r} 0000000000000000 = 0 = \text{top number} \\ 1111111111111111 = \text{bottom number with all its bits flipped} \\ + \qquad \qquad \qquad 1 = \text{carry in on rightmost full adder} \\ \hline 0000000000000000 \end{array}$$

Sign of true result

- Needed to determine the result of a signed comparison.
- The **S** flag has the sign of the true result of a two's complement addition only if overflow does **NOT** occur.
- To get true sign, use **S XOR V**.

Getting the sign of the true result

$$\text{data XOR } 0 = \text{data}$$

and

$$\text{data XOR } 1 = \overline{\text{data}}$$

Using this relationship, let's determine what we get when we apply S and V to an XOR gate. If $V = 0$ (i.e., if signed overflow does not occur) then

$$S \text{ XOR } V = S \text{ XOR } 0 = S = \text{sign of true result}$$

But for this case, S , the sign of the computed result, is also the sign of the true result. Now let's consider the case when $V = 1$ (i.e., signed overflow occurs), then

$$S \text{ XOR } V = S \text{ XOR } 1 = \bar{S} = \text{sign of true result}$$

Scientific notation

Significand \times power of 10

where significand is the number in which the decimal point is to the immediate right of the leftmost non-zero digit.

154 in scientific notation is
 1.54×10^2

Binary scientific notation

$$1110.11 = 1.11011 \times 2^3$$

Floating point format for

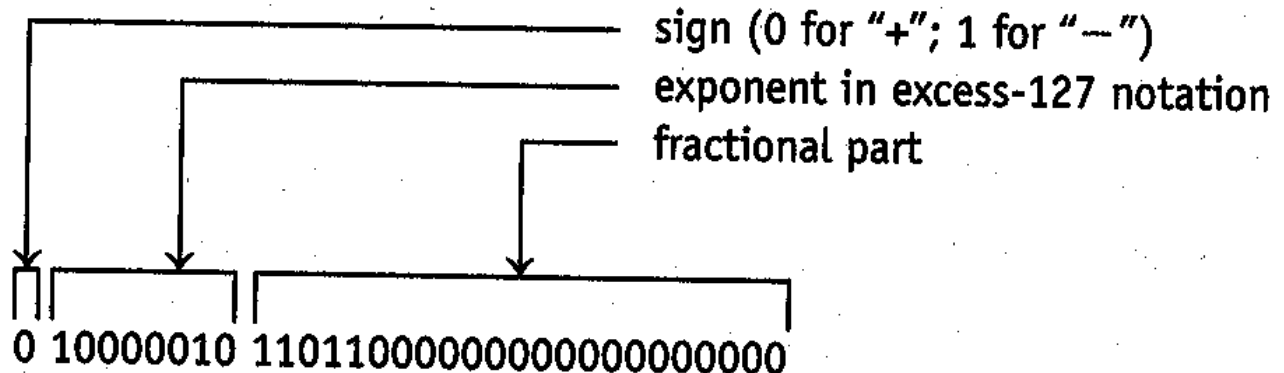
$$+1.11011 \times 2^3$$

- Its sign is +.
- 1.11011 is called the *significand*
- .11011 is called the *fractional part*
- 3 is the *exponent*
- In floating-point format, store the sign (0 for +, 1 for -), the exponent in excess-127, and the fractional part.

Floating-point format for 1.11011×2^3

Note that the bit to the left of the binary point does not appear in the floating point format. It is called the *hidden bit*.

FIGURE 1.16



Normalization

Shifting a number so that the binary point is in the correct place for floating-point format (to the immediate right of the leftmost 1).

Special values

FIGURE 1.17

Exponent	Significand Field	Value
all zeros	zero	zero
all ones	zero	∞ (infinity)
all ones	nonzero	NaN (Not a Number)
all zeros	nonzero	denormal number

Denormal number

Exponent = -126 and hidden bit = 0

0 00000000 000000000000000000000001

represent -126 and 0.000000000000000000000001, respectively. Thus, its value is

$$0.000000000000000000000001 \times 2^{-126}$$

Computational error

- Floating-point computations are not exact.
- Errors can grow to the point where the final results of a computation may be meaningless.

RULE 1 Not all numbers can be represented exactly as floating-point numbers.

RULE 2 Errors can accumulate.

The sum computed by the first program on the next slide is *not* 1.

FIGURE 1.18 a)

```
1 class E1 {
2     public static void main(String[] s)
3     {
4         float sum = 0.0f, z = 0.001f;
5         for (int i = 1; i <= 1000; i++)
6             sum = sum + z;
7         System.out.print("sum = ");
8         System.out.println(sum);
9     }
10 }
```

b)

```
1 class E2 {
2     public static void main(String[] s)
3     {
4         float sum = 0.0f, z = 1.0f/1024.0f;
5         for (int i = 1; i <= 1024; i++)
6             sum = sum + z;
7         System.out.print("sum = ");
8         System.out.println(sum);
9     }
10 }
```

RULE 3 It is dangerous to test for the equality of floating-point numbers.

The first program on the next slide is an infinite loop.

FIGURE 1.19 a)

```
1 class E3 {  
2     public static void main(String[] s)  
3     {  
4         float x = 0.0f;  
5         while (x != 1.0f)    // infinite loop  
6             x = x + 0.1f;  
7         System.out.print("x =");  
8         System.out.println(x);  
9     }  
10 }
```

b)

```
1 class E4 {  
2     public static void main(String[] s)  
3     {  
4         float x = 0.0f;  
5         while (Math.abs(x - 1.0f) > 0.00001f)  
6             x = x + 0.1f;  
7         System.out.print("x = ");  
8         System.out.println(x);  
9     }  
10 }
```

Round-off error

When we perform a computation with floating-point numbers, not all the bits of the fractional part of the result may fit into the 23-bit significand field of a floating-point number. For example, suppose we want to add the floating-point numbers whose values are

$$1.100000000000000000000000 \times 2^{30} \quad (\text{1st number})$$

and

$$1.101010000000000000000000 \times 2^{10} \quad (\text{2nd number})$$

Before we can add, we have to shift the binary point in the second number 20 positions to the left to increase its exponent to 30. We can then add:

$$\begin{array}{rcl} 1.100000000000000000000000 & \times 2^{30} & \text{1st number} \\ + 0.00000000000000000000110101 & \times 2^{30} & \text{2nd number (unnormalized)} \\ \hline 1.10000000000000000000110101 & \times 2^{30} & \\ \quad \quad \quad \uparrow & & \text{these bits are lost} \end{array}$$

Guard	Round	Sticky	Action	Effect
1	0	1	} Add 1 to 23rd bit, then chop off extra bits.	Round away from zero.
1	1	0		
1	1	1		
0	0	0	} Chop off extra bits.	Round toward zero.
0	0	1		
0	1	0		
0	1	1		
1	0	0	} Add 1 to 23rd bit, but only if it is 1. Then chop off extra bits.	Depends on 23rd bit, which always ends up 0.

RULE 4 If the magnitude of the floating-point number L is sufficiently larger than the magnitude of the floating-point number S , then $L + S = L$.

The program on the next slide displays “equal”.

FIGURE 1.21

```
1 class E5 {  
2     public static void main(String[] s)  
3     {  
4         float L, S;  
5         L = 1.0f;           // L is a Large number  
6         S = 1E-30f;        // S is a Small number  
7         if (L + S == L)  
8             System.out.println("equal");  
9         else  
10            System.out.println("not equal");  
11     }  
12 }
```


Associative operator

Order of performing operations
does not matter.

$$(a + b) + c = a + (b + c)$$

RULE 5 Floating-point addition is not associative.

The programs on the next slide
output different sums.

FIGURE 1.22

a)

```
1 class E6 {  
2     public static void main(String[] s)  
3     {  
4         float sum = 0.0f;  
5         for (int i = 1; i <= 100; i++)  
6             sum = sum + 1.0f / i;  
7         System.out.println(sum);  
8     }  
9 }
```

b)

```
1 class E7 {  
2     public static void main(String[] s)  
3     {  
4         float sum = 0.0f;  
5         for (int i = 100; i >= 1; i--)  
6             sum = sum + 1.0f / i;  
7         System.out.println(sum);  
8     }  
9 }
```

RULE 6 Subtracting nearly equal values can produce meaningless results.

Suppose x and y are true values,
and X and Y , respectively, are their
computed values that include small
errors.

$$X = x + 0.0001 \quad Y = y - 0.0001$$

Now compute $X - Y$:

$$X - Y = x + 0.0001 - (y - 0.0001) = \\ (x - y) + 0.0002$$

The absolute error (0.0002) will be large relative to the true result ($x - y$) if x and y are nearly equal.